Mitigating Dual Load Imbalance via Dynamic Cooperative Scheduling in Distributed Key-Value Stores

Jiakun Zhang*, Patrick P. C. Lee†, Wenzhe Zhu*, Yongkun Li*‡, Shuyi Zhang*, Yinlong Xu*‡
* University of Science and Technology of China † The Chinese University of Hong Kong
‡ Anhui Province Key Laboratory of High Performance Computing, USTC

Abstract—**Distributed key-value (KV) stores are essential components of modern computing infrastructure, enabling efficient storage and management of large-scale datasets. Existing distributed KV stores often shard data by key ranges into multiple regions and distribute the regions across multiple nodes. However, such range-based sharding leads to load imbalance in two dimensions: CPU utilization and disk I/O bandwidth. Our analysis reveals that the two dimensions exhibit misaligned and dynamic behaviors. Moreover, their tight coupling, where scheduling one dimension affects the other, makes it more challenging to simultaneously achieve balance in both. To address this challenge, which we term dual load imbalance, we propose Libra, a cooperative scheduling framework that monitors the interactions of CPU and disk I/O loads and carefully migrates regions across nodes based on the critical dimension. We implement Libra atop TiKV, a production distributed KV store, and show that Libra increases throughput by up to 72.1% and reduces tail latency by up to 56.7% compared to state-of-the-art approaches.**

I. Introduction

Distributed key-value (KV) stores enhance the performance, scalability, and reliability of local (single-node) KV stores by organizing data as *KV pairs* and distributing them across multiple *nodes*. They are integral to applications such as NoSQL/NewSQL databases [4], [28], [38], [55], [58], file systems [37], search engines [11], and machine learning [12]. For scalable range queries, distributed KV stores often employ *range-based sharding* [4], [11], [14], [52], [58], which divides an ordered key space into contiguous *regions* and assigns them to nodes, each of which manages multiple ranges of KV pairs.

Load imbalance is a common issue in distributed KV stores [26], [36], [60], where some nodes experience disproportionately higher request traffic than others and degrade overall system performance. Under range-based sharding, load imbalance can be caused by uneven distributions of KV pairs [28], varying access patterns [5], [9], and internal storage management for index structures and regions [28], [40]. At the hardware level, our measurement (§II-B) reveals that load imbalance often manifests as bottlenecks in two dimensions, namely *CPU utilization* and *disk I/O bandwidth*, each of which exerts a different impact on system performance.

We observe that loads in these two dimensions do not always align with request traffic, but instead exhibit complex, unique characteristics, which we term *dual load imbalance*. First, CPU and disk I/O load distributions are often misaligned across nodes; for example, a node with high CPU utilization may have low disk I/O demands, and vice versa. In addition to uneven request distributions [5], [9], this misalignment is primarily caused by internal characteristics of KV stores, such as variations in KV pair sizes [9] and differing resource demands between read and write operations [28], [40]. This complicates load balancing, as optimizing one dimension may adversely affect load balancing in the other. Second, both dimensions often show significant, simultaneous load imbalance, with distributions shifting dynamically over time.

Effective load balancing in distributed KV stores requires careful scheduling of both CPU and disk I/O resources. Existing distributed KV stores commonly employ *migration* [8], [36], [52] to dynamically relocate regions across nodes for load balancing. However, achieving load balancing is challenging due to the complex interplay between CPU and disk I/O. Existing migration approaches struggle to effectively address the two-dimensional complexity. They adapt existing single-dimensional migration techniques to two-dimensional scenarios using either parallel and isolated execution (referred to as ISO) or weighted aggregation (referred to as WTD). ISO balances each dimension independently, leading to inter-dimensional interference; alternatively, WTD combines CPU and disk I/O loads into a single weighted score for load balancing, but relies on static weights for all nodes and regions and overlooks the misalignment of both dimensions. Our experiments (§II-C) show that even with ISO or WTD, the load on the most heavily loaded node can still exceed the cluster average by more than 50% in distributed KV stores. This motivates the need for a *coordinated*, real-time load balancing solution.

We propose Libra, a cooperative scheduling framework that achieves simultaneous load balancing of both CPU and disk I/O in distributed KV stores based on range-based sharding. Libra monitors real-time CPU and disk I/O usage for each region to capture their dynamic and complex correlations. It identifies the *critical dimension* between the two dimensions by comparing their deviations from average usage across all nodes. It also updates node resource usage immediately after each scheduling operation, and iteratively performs scheduling operations to guide both dimensions toward a balanced state.

To ensure practical efficiency of Libra, we propose three novel techniques: (i) a sufficient condition for two-dimensional resource usage per region that ensures feasible load balancing via migration, coupled with an efficient region-splitting approach for achieving the sufficient condition; (ii) a lightweight synchronization scheme that enables the centralized manager to track real-time region loads; and (iii) a lazy scheduling approach that mitigates migration overhead. Our contributions are summarized as follows.

- We analyze the dual load imbalance problem and the complex correlation between CPU and disk I/O. We also analyze the limitations of three commonly used load balancing approaches in distributed KV stores.
- We propose a two-dimensional cooperative scheduling framework that jointly manages both CPU and disk I/O loads, so as to simultaneously achieve load balancing in both dimensions.
- We implement Libra atop TiKV [52], a production distributed KV store. Our evaluation shows that Libra increases throughput by up to 72.1% and reduces tail latency by up to 56.7% compared to baseline approaches. Our source code is now open-sourced at **https://github.com/JK1Zhang/Libra**.

II. Background and Motivation

A. Distributed KV Stores

We first provide an overview of the distributed KV stores considered in this paper, as shown in Figure 1.

Data sharding. Practical distributed KV stores commonly employ range-based sharding to partition KV pairs across nodes [1], [4], [11], [14], [52], [58]. It divides the entire key space into non-overlapping regions, each of which typically has a size from tens to hundreds of megabytes [4], [11], [52]. When a region reaches its capacity threshold, it is *split* into multiple smaller regions to maintain manageable sizes. A dedicated *manager* node is deployed for handling region management tasks, such as assigning region IDs, storing metadata, and determining region placement. In this paper, we consider a distributed KV store with n nodes (N_1, N_2, $\cdots$, N_n) and m regions (R_1, R_2, $\cdots$, R_m). For fault tolerance, distributed KV stores employ *replication* [4], [11], [13], [18], [20], [38], [52] by distributing multiple replicas of a region across nodes. They also employ a consensus protocol, such as Paxos [11] or Raft [28], to select a leader and ensure consistency among replicas.

Another commonly used data sharding strategy is *hash-based sharding* [2], [38], [55], which maps KV pairs to nodes using hashing algorithms (e.g., consistent hashing [34]). Hash-based sharding allows clients to deterministically locate the node for a KV pair via hashing without a dedicated manager, yet it incurs high overhead in range queries [53]. In this paper, we focus on range-based sharding.

Storage engine. Each node manages KV pairs for different regions using a unified storage engine. A mainstream choice of the storage engine is the *log-structured merge tree (LSM-tree)* [50], which organizes KV pairs in a hierarchical, multi-level tree structure [4], [11], [22], [25], [38], [52]. New KV pairs are appended to the lowest level, and moved from lower to higher levels via *compaction*. Each level (except the lowest level) maintains sorted KV pairs, with overlapping key ranges across different levels. To read a KV pair, the LSM-tree searches from the lowest to higher levels until the KV pair is located.

Request handling. To access or insert a KV pair, the client queries the manager to identify the node that manages the relevant region for the key and sends a request to it for the KV pair. Requests are typically implemented via remote procedure

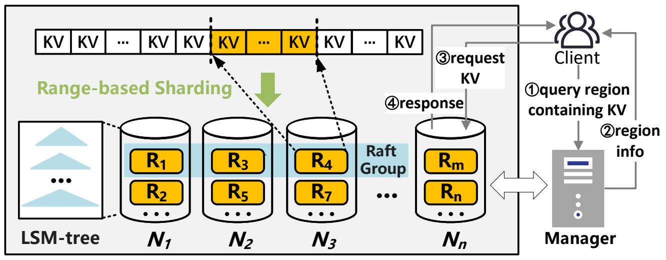


Figure 1: Architecture of a distributed KV store.

calls (RPCs) and are directed to the node for the leader region among replicas to ensure consistency. For read requests, the node retrieves and returns the KV pair. For write requests, the node for the leader region synchronizes modifications across all replicas before responding to the client.

B. Dual Load Imbalance in CPU and Disk I/O

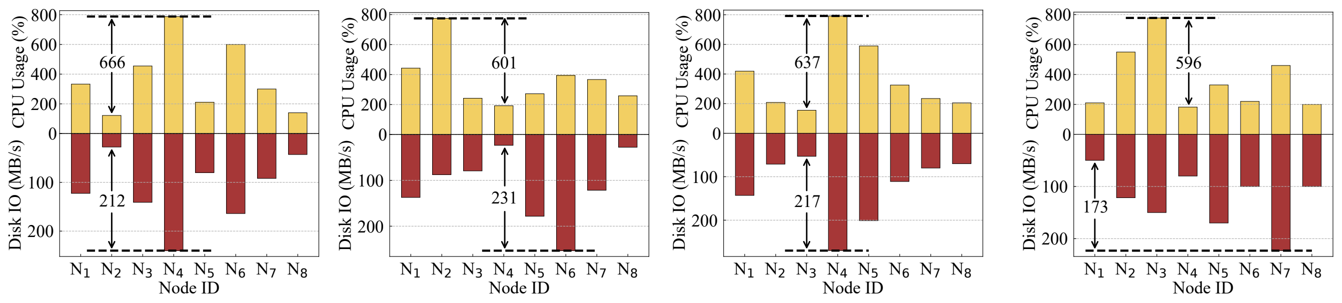
Load imbalance is a critical challenge to distributed KV stores, where CPU and disk I/O are primary resource bottlenecks. Prior studies identify CPU exhaustion [39], [61], [65] and disk I/O bandwidth limitations [6], [19], [46], [54], [63] as the root causes of high access latencies in KV storage. In contrast, memory and network bandwidth resources are often abundant and will not be the bottleneck. For example, in Bigtable deployment [11] with 400 GB of data, only 1 GB of memory is used, and 8,069 busy servers have no more than 17 GB/s of peak network throughput. Thus, we focus on CPU and disk I/O to address load imbalance in distributed KV stores.

Note that our scheduling design incurs extra network overhead. For example, we need to collect load status from various nodes, but such overhead is limited compared to Gbps-level network bandwidth. Also, hot regions generally make up only a small portion of the entire system workloads [5], so there is only limited data migrated per node for load balancing (with less than a few hundred MB). Furthermore, we design a protective solution (§III-B) to control transmission costs.

We define *dual load imbalance* as the simultaneous load imbalance in CPU and disk I/O. To quantify the severity of dual load imbalance, we conduct experiments using production workloads. We consider a TiKV cluster (based on range-based sharding) and a Cassandra cluster (also configured with range-based sharding) with default configurations (see §V-A for testbed details). We distribute KV pairs evenly across eight nodes and disable scheduling in our evaluation. We configure a client to send a steady stream of write requests following the Mixgraph benchmark derived from Facebook's production workloads [9]. Note that other workloads (e.g., YCSB core workloads [7]) also exhibit similar dual load imbalance.

We consider two KV size distributions in our evaluation: fixed-size (1 KB in Figures 2(a) and 2(c)) and Pareto-distributed (average 1 KB in Figures 2(b) and 2(d)) [21], [43]. Fixed-size KV pairs serve as a baseline, as they are uncommon in real-world workloads [21].

We consider two load metrics per node: (i) *CPU utilization*, measured by the sum of per-core utilizations across eight cores (the maximum CPU utilization is 800%), and (ii) *disk*



(a) TiKV with fixed KV size (b) TiKV with skewed KV size (c) Cassandra with fixed KV size (d) Cassandra with skewed KV size

Figure 2: Distribution of CPU usage and disk I/O across the cluster.

throughput, measured by the amount of I/Os in megabytes per second (the bandwidth of SATA SSD here is approximately 300 MB/s, which is read-write-mixed due to compaction reads). The cluster runs for 10 minutes, with metrics sampled every 15 s. Figure 2 shows the average results, with two key observations.

Observation #1: Dynamic CPU and disk I/O imbalance. Both CPU and disk I/O loads vary significantly across nodes. Highly loaded nodes experience substantially higher load demands than lightly loaded nodes. For example, in Figure 2(b), the CPU utilization of N_2 is about $4\times$ that of N_4, while the disk throughput of N_6 is about $11\times$ that of N_4.

Observation #2: Misaligned CPU and disk I/O loads. CPU and disk I/O loads are often misaligned across different nodes. High CPU utilization is not necessarily correlated with high disk throughput, and vice versa. For instance, in Figure 2(b), N_2 exhibits the highest CPU utilization but relatively low disk throughput. Our further analysis reveals that this misalignment is caused by numerous small-size KV requests issued to N_2 (averaging about 300 bytes). This misalignment suggests that balancing one resource dimension alone may not address load imbalance in both dimensions. Notably, other factors, such as differences in resource demands between read and write requests, also contribute to load misalignment.

C. Limitations of Existing Scheduling Approaches

Data migration is a common strategy for load balancing in mainstream distributed KV stores [4], [8], [14], [28], [58]. It identifies hot data on overloaded nodes and relocates it to lightly loaded nodes. Existing scheduling implementations for migration can be categorized as follows:

- *Single-dimensional scheduling.* This approach monitors only a single dimension of nodes, either CPU utilization (referred to as "1D-CPU") or disk I/O throughput (referred to as "1D-IO"), and migrates hot regions from the most loaded node to the least loaded node. It follows a *greedy* approach to select the hottest region in the most loaded node for migration.
- *Isolated scheduling.* This approach (referred to as "2D-ISO") is adopted by TiKV [52] and CockroachDB [58]. It monitors CPU utilization and I/O loads *independently* and triggers region migration based on the greedy approach when either dimension becomes highly loaded, without considering the possible influence on the other dimension.
- *Weighted scheduling.* This approach (referred to as "2D-WTD") is adopted by HBase [4]. It combines both CPU and I/O loads into a single score with fixed weights, such that a higher score indicates a more loaded node. It can be viewed as score-based single-dimensional scheduling.

Limitations. Given that both CPU and I/O loads of a node are closely related to locally stored regions, migrating regions to balance the one resource dimension (e.g., CPU utilization) can interfere with the other dimension (e.g., disk I/O throughput). Such interference, together with the misaligned CPU and I/O loads, renders existing scheduling strategies inefficient. We verify this observation by conducting an evaluation on the same eight-node TiKV cluster described in §II-B. In particular, we use a 0.5:0.5 weight ratio for 2D-WTD.

Notably, existing approaches typically balance loads based on software-level metrics and do not support direct scheduling at the hardware resource level. Since it is difficult to assess the hardware resource usage of individual regions (a region is only a logical unit and all regions on a node are managed collectively by the storage engine), systems typically only monitor hardware usage at the node level. To explore the effectiveness of different scheduling policies in hardware scheduling, we leverage a technique (§IV-A) that enables scheduling based on hardware metrics of regions.

Figure 3 shows the CPU utilization and disk throughput across all nodes in the cluster for each scheduling strategy. We make the following observations. First, for single-dimensional scheduling, while 1D-CPU or 1D-IO balances its targeted dimension, the other dimension remains highly imbalanced due to misaligned CPU and disk I/O loads. Second, for isolated scheduling, 2D-ISO improves load balancing in both dimensions, but still fails to achieve complete load balancing due to the interference between CPU and disk I/O. Migration for addressing one dimension (e.g., reducing CPU load on N_2) can inadvertently disrupt the other (e.g., increasing disk throughput on N_6). Finally, for weighted scheduling, 2D-WTD slightly outperforms 2D-ISO in reducing the gap between the highest and lowest loads, but remains ineffective. The reason is that both CPU and I/O loads are assigned equal weights across all nodes in load balancing. It fails to capture the misaligned nature of the CPU and I/O loads. For example, if CPU and disk I/O loads are measured by a score from 0 to 100 with an

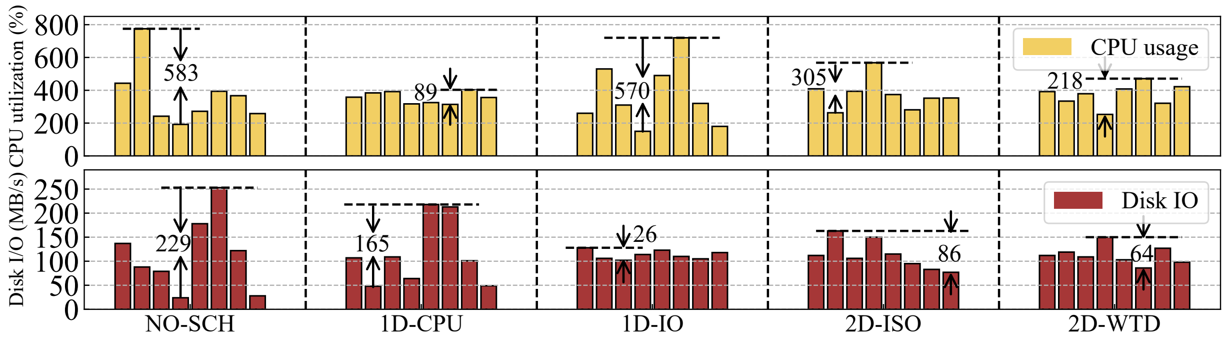


Figure 3: Impact of no scheduling ("NO-SCH"), single-dimensional scheduling based on either CPU or disk I/O ("1D-CPU" and "1D-IO"), isolated scheduling ("2D-ISO"), and weighted scheduling ("2D-WTD"). The bars in each sub-figure correspond to the loads in nodes $N_1 - N_8$.

equal weight of 0.5, a node with high CPU load (100) but low disk I/O load (50) and another node with low CPU load (50) but high disk I/O load (100) both yield a score of 75. This creates an illusion of load balancing, but the actual CPU and disk I/O loads remain significantly imbalanced.

III. Cooperative Scheduling

We propose a novel scheduling framework called *cooperative scheduling* to address dual load imbalance.

A. Problem Formulation

We define the notation for the dual load balancing problem as follows. Consider a distributed KV store with m regions R_1, $R_2, \cdots, R_m$ and n nodes $N_1, N_2, \cdots, N_n$. We represent the load of each region R_i as a two-dimensional load vector (α_i, β_i), in which α_i and β_i represent the CPU utilization and disk I/O bandwidth of R_i, respectively. Similarly, we represent the load of each node N_j as a two-dimensional load vector (A_j, B_j), where $A_j = \sum \alpha_i$ and $B_j = \sum \beta_i$ aggregate the CPU utilization and disk I/O bandwidth of all regions in node N_j, respectively.

To allow comparisons across CPU and disk I/O loads, we normalize the load vectors as follows:

$$A'_j = A_j/\overline{A}, \quad B'_j = B_j/\overline{B} \qquad j \in [1,n] \tag{1}$$

$$\alpha'_i = \alpha_i/\overline{A}, \quad \beta'_i = \beta_i/\overline{B} \qquad i \in [1,m], \tag{2}$$

where $\overline{A}$ and $\overline{B}$ denote the average CPU utilization and disk I/O across all n nodes, respectively. Based on normalization, we can directly compare the CPU and disk I/O loads of a region or a node. For example, if α'_i exceeds β'_i for region R_i, then R_i imposes higher normalized CPU load than normalized disk I/O load, indicating that it is now CPU-intensive.

We quantify the deviation of each node's resource utilization with respect to the system average over all nodes.

$$\delta_j = \max\left(\frac{A_j - \overline{A}}{\overline{A}}, \frac{B_j - \overline{B}}{\overline{B}}\right) = \max(A'_j - 1, B'_j - 1) \quad j \in [1,n] \tag{3}$$

We further define the load imbalance degree Δ, which specifies the highest resource utilization deviation of all nodes.

$$\Delta = max(\delta_j) \quad j \in [1,n] \tag{4}$$

A smaller Δ indicates better load balance; ideally, we aim for $\Delta = 0$, where the loads of all nodes have no deviation from the system average in both dimensions. Our scheduling objective is to constrain Δ below a predefined threshold λ, where λ indicates the maximum acceptable deviation from the system average. The unified λ assumes all nodes are homogeneous and cannot adapt to heterogeneous nodes. λ is a configurable parameter. In general, a small value of λ is preferred to ensure low latency and meet SLA requirements. However, in practice, λ cannot be arbitrarily small, as its precision is limited by load estimation discrepancy (§IV-A) and fluctuations. Also, as λ decreases, the scheduling overhead increases due to an increasing number of migrations and splitting operations and extra CPU costs of managing numerous smaller regions (see details in §V-F).

B. Cooperative Scheduling Framework

Cooperative scheduling is activated when Δ exceeds λ. It then triggers migrations to reduce Δ to an acceptable level. Unlike isolated or weighted scheduling approaches, which are ineffective due to load misalignment (§II-C), cooperative scheduling *iteratively* evaluates the system load state and executes migration based on the most recent load information in periodic statistics and measured upload performance from data nodes, so as to ensure adaptive and effective load balancing.

Figure 4 depicts the idea of cooperative scheduling, which iteratively performs two key steps: (i) selecting the critical dimension and (ii) selecting a region for migration.

Selecting the critical dimension. In each iteration, the cooperative scheduling framework selects the *critical dimension*, either CPU utilization load or disk I/O load, that contributes most significantly to the current load imbalance. Intuitively, performing load balancing on the critical dimension has a higher load balancing gain than the other dimension. Specifically, the framework identifies the node, N_c, with the highest deviation δ_j across all nodes N_j, where $j = 1, 2, ..., n$. Given N_c, we select the critical dimension by comparing the normalized CPU load and the normalized disk I/O and check which dimension dictates δ_c: if $A'_c > B'_c$, the CPU utilization load is the critical dimension; otherwise, the disk I/O load is the critical dimension.

The approach is intuitively sound by directly comparing A'_c and B'_c. It prioritizes the dimension with the global peak load

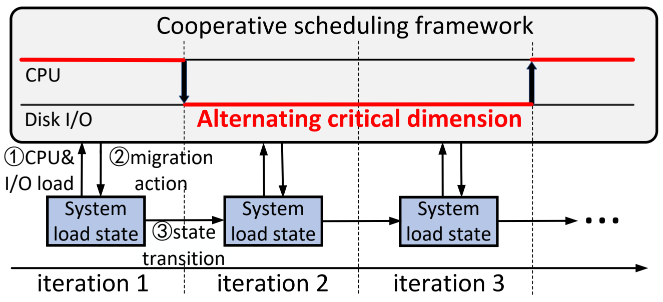


Figure 4: Cooperative scheduling framework.

(i.e., the highest load of all nodes across both dimensions), thereby maximizing load balancing gains while minimizing interference with the non-critical dimension. Also, it simplifies region selection, as we can focus on identifying the highly loaded regions only in the most overloaded node.

Selecting the region for migration. Based on the critical dimension, the cooperative scheduling framework selects a region for migration. For clarity, we assume that in the current iteration, the critical dimension is the CPU load; the discussion is analogous for disk I/O load, with the CPU and I/O load parameters swapped.

We first select the *source node*, N_{src}, with the highest CPU load, from which a region will be chosen for migration. We also select the *destination node*, N_{dst}, with the lowest CPU load, to which the region will be migrated.

We select an appropriate region from N_{src} for migration. We sort the regions in N_{src} by their CPU load in descending order. For each region R_i, we determine if R_i is selected based on the following three conditions:

- We check the load constraint by determining whether $A'_{dst} + \alpha'_i < 1 + \lambda$. This ensures that if R_i is migrated to N_{dst}, the CPU load of N_{dst} will not exceed the average load of all nodes plus a tolerable deviation $\lambda * \overline{A}$.
- We check the interference constraint by determining whether $\beta'_i < \alpha'_i$. This condition limits the interference to the non-critical I/O dimension by ensuring that the I/O load of R_i is proportionally lower than its CPU load. For example, in the ideal situation where $\beta'_i = 0$, migrating R_i will not affect the I/O load of nodes. Thus, there is no interference between CPU and I/O load dimensions.
- We check the benefit threshold by determining whether $\alpha'_i > \frac{1}{500}$, an empirically measured threshold, to ensure that the gain of migrating R_i outweighs the overhead. A larger α'_i implies less extra network overhead incurred by scheduling.

If all conditions are met, R_i is selected for migration; otherwise, we continue to search for another appropriate region. If no region satisfies the conditions after all regions in N_{src} have been examined, the current round of selection terminates, and the framework proceeds with either of the following options: splitting the current region, or moving on to the next node. The whole scheduling process terminates when either load balance is reached or no suitable region can be found.

C. Design Issues

Integrating the cooperative scheduling framework into a distributed KV store poses three key challenges.

Issue #1: System load monitoring. Accurately attributing resource loads to individual regions is difficult, as load monitoring typically occurs at the node or process level. In particular, cooperative scheduling requires precise, timely load data, but frequent monitoring increases communication overhead and can overwhelm the manager, which is responsible for collecting load information. A lightweight monitoring module is needed to capture region-specific loads efficiently.

Issue #2: Precise region splitting. The cooperative scheduling framework selects regions for migration based on the three conditions described in §III-B, but large regions are coarse-grained and may fail to meet these criteria. This causes the algorithm to terminate easily, failing to achieve load balancing. Splitting large regions is a potential solution, but dividing them into precise load sizes is complicated due to the varying popularity of KV pairs. Also, excessively small regions incur significant management overhead. A region-splitting module should support cooperative scheduling by creating regions with an appropriate load size.

Issue #3: Scheduling plan optimization. Cooperative scheduling is an iterative algorithm that executes migrations sequentially, but its efficiency can be improved by queuing actions for collective optimization. Analyzing collectively multiple migration actions enables the elimination of redundant migrations and splits, thereby reducing CPU cycles, data transfers, and disk I/O overhead associated with region management.

IV. Libra Design

Libra addresses the challenges in §III-C via three core techniques. First, it integrates a *lightweight system monitoring* approach (§IV-A) to track resource usage for each region, using a metric mapping technique that mixes high- and low-precision statistics to monitor the system state without overloading the centralized manager. Second, it develops a *precise region-splitting* approach (§IV-B) to calculate an appropriate upper bound for region loads and determine the optimal split thresholds for regions. Finally, it uses a *lazy scheduling* strategy (§IV-C) to defer execution of splitting and migration until necessary by grouping and analyzing multiple actions in a pending queue to mitigate scheduling overhead.

A. System Load Monitoring

Libra employs two lightweight techniques to efficiently monitor hardware resource usage: a *metric mapping* scheme for estimating region-level loads of the two dimensions and a *two-layer information gathering* framework for collecting load information with limited communication overhead.

Load estimation by metric mapping. Measuring CPU and disk I/O loads directly for regions is challenging. Libra adopts a *metric mapping* scheme to correlate software-level metrics, such as operations per second (OPS), with hardware-level resource usage, so as to enable accurate load estimation with minimal overhead. It introduces a statistics module in each node to

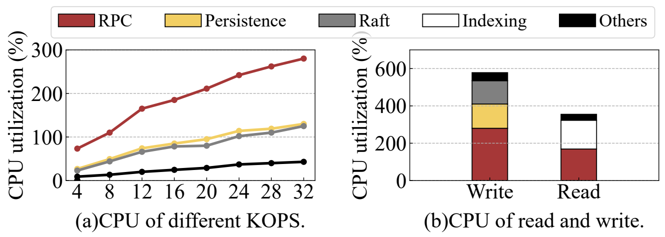


Figure 5: Relationship between requests and CPU usage.

collect software-level metrics, which are mapped to hardware-level metrics based on the established relationships.

We first study the correlation between OPS and CPU usage. Through our analysis in §II-B, we identify four major components for CPU usage: (i) Remote Procedure Call (RPC) handling of requests, (ii) Raft message processing, (iii) data persistence, and (iv) KV indexing. We measure the CPU usage of the threads associated with each component. Figure 5 shows the results for the TiKV cluster (the Cassandra cluster shows similar behaviors).

Figure 5(a) shows a positive correlation between OPS and CPU usage for write operations (note that writes in the KV engine incur no indexing cost). The CPU load of region R_i on node N_j can be roughly estimated as:

$$\alpha_i = A_j * OPS_{R_i}/OPS_{N_j},$$

where A_j is the measured CPU utilization. However, this estimation does not address both write and read operations and their differences in resource usage.

Figure 5(b) shows that write requests consume 63% more CPU than read requests at 32 KOPS. To account for this difference in CPU usage, we separate the throughput of read and write operations and denote them by *ROPS* and *WOPS*, respectively. We introduce a *correction factor* c to represent the ratio of CPU usage between write and read operations; from our empirical measurement, we set $c = 1.63$. Thus, the *refined* CPU load of R_i on N_j is estimated as:

$$\alpha_i = A_j * (ROPS_{R_i} + c * WOPS_{R_i})/(ROPS_{N_j} + c * WOPS_{N_j}).$$

In long-running systems, the correction factor remains relatively stable, as it is primarily influenced by the underlying hardware. In our experiments, the fluctuations of the value of c are always within 10%. The correction factor can be measured via offline profiling, either before system startup or when substantial changes occur (e.g., significant changes in the number of nodes, total data volume, or hardware configuration).

We can also establish a similar relationship between disk I/O and user I/O (i.e., the amount of user-requested data per second). As in CPU estimation, it is necessary to distinguish between reads and writes. Read requests often leverage caching to absorb disk I/O, while write requests incur significantly higher disk I/O due to LSM-tree compaction and replica synchronization [5], [9]. Thus, we directly measure region loads and take into account the impact of replication for accurate I/O estimation.

Two-layer information gathering. To accurately characterize and partition a region's internal load, detailed request information is required beyond basic statistics. Centralized data collection would incur significant communication overhead (§IV-B). Libra employs a two-layer information gathering framework, as shown in Figure 6, which mitigates communication overhead between the manager and cluster nodes. It divides the information gathering process into two layers: each node locally collects detailed request information, while the manager processes aggregated summaries to reduce data transmission.

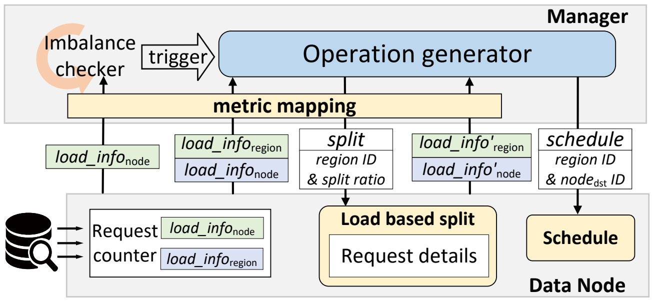


Figure 6: Two-layer information gathering.

Each node in the cluster computes two load metrics: region-level (*load_info*$_{region}$) and node-level (*load_info*$_{node}$). These metrics are batched into lightweight heartbeats that are of a few bytes each, and are periodically sent to the manager. The manager runs an *imbalance checker* to monitor *load_info*$_{node}$ for any load imbalance. If imbalance is detected and region splitting is required, the manager selects a large region based on *load_info*$_{region}$ and sends the corresponding node (say N) a split operation request, including the region ID and split ratio. Node N then collects detailed request information for the specified region over a defined period and executes the split operation (§IV-B). Once splitting is complete, the manager generates a scheduling plan using updated heartbeats (*load_info'* in Figure 6) and sends it to N. Finally, N schedules the region for migration based on the region ID and destination node ID.

By delegating split decisions to individual nodes, the two-layer framework significantly reduces communication overhead. Given the high request volume in distributed KV stores (e.g., up to hundreds of thousands of requests per second), centralized decision-making would substantially increase the number of heartbeats. The two-layer framework allows independent heartbeat rates for *load_info*$_{node}$ and *load_info*$_{region}$, enabling trade-offs between load balancing timeliness and information gathering overhead. Note that high-frequency heartbeats are not necessarily more efficient, as they are susceptible to temporary load changes. Based on empirical measurement, we set the default interval to 10 s to filter short-term fluctuations. If significant fluctuations are detected in recent heartbeats (when the load gap between consecutive intervals exceeds a 20% threshold), Libra temporarily suspends scheduling to prevent erroneous decisions and unnecessary data migration. It resumes scheduling only when the load gap drops below the threshold.

B. *Load-based Splitting*

Libra supports *load-based splitting*, a technique that splits regions based on a specified load ratio, independent of their

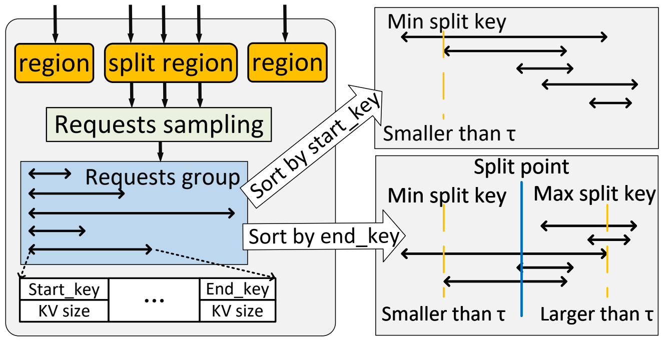


Figure 7: Load-based splitting.

internal load distribution. Unlike size-based splitting, which splits regions based on the region sizes and is sensitive to load skewness, load-based splitting ensures balanced load distribution. Figure 7 depicts the idea of load-based splitting. When a region requires splitting, the node begins to sample requests for the region over a period of time. It identifies the optimal load split point, and splits the region accordingly. The load-based splitting approach comprises three main components.

Upper bound of region load. As stated in Issue #2 (§III-C), excessively large regions can lead to inefficient scheduling, while overly small regions result in additional overhead. To address this issue, we prove that Libra should cap region loads at an upper bound $\lambda/2$ (recall that λ represents the upper bound of the load imbalance degree; see §III-A), so as to ensure the balanced state defined in §III-A.

Theorem 1. *If the normalized loads α_i' and β_i' of each region R_i do not exceed $\lambda/2$, the system can achieve the load balance state (i.e., $\Delta \leq \lambda$) by scheduling.*

Proof. Suppose that the both region loads α_i' and β_i' of each region R_i are no greater than λ_2. We classify regions as *A-class* if $\alpha_i' > \beta_i'$ (i.e., CPU-dominant) and *B-class* if $\alpha_i' \leq \beta_i'$ (i.e., I/O-dominant). For an empty node N_j, we start by placing an A-class region, followed by placing B-class regions on N_j until N_j's disk I/O load B_j' exceeds N_j's CPU load A_j'. We next place A-class regions on N_j until A_j' exceeds B_j'. We repeat this cycle until both A_j' and B_j' exceed one. We repeat the process for other nodes.

We argue that Libra achieves a balanced state based on the above placement strategy. Note that each time we add a region to a node, both CPU and I/O loads of the node will increase, but by no more than $\lambda/2$ based on our assumption. Thus, A_j' and B_j' always differ by no more than $\lambda/2$. When both A_j' and B_j' exceed one, the lower load reaches at most $1+\lambda/2$, while the higher load reaches up to $1+\lambda$. Thus, we have $\delta_j \leq \lambda$, and hence $\Delta \leq \lambda$. $\square$

Based on Theorem 1, we split a region only when either of its normalized CPU or disk I/O loads exceeds $\lambda/2$. When the loads of the resulting regions are below $\lambda/2$, there will be no further splitting. Note that this condition is sufficient but not necessary, and Libra performs splitting only when no lighter-loaded regions meet the migration criteria in §III-B.

Analysis of convergence. Based on the above constraints, we analyze the convergence of Libra's cooperative scheduling (notation follows §III). If $\Delta > \lambda$, there exists at least one node with $\delta_j > \lambda$ (i.e., $\max(A_j', B_j') > 1+\lambda$). We classify such nodes based on the difference $d_j = |A_j' - B_j'|$.

- **Case 1:** $d_j < \lambda/2$**.** If a node has $d_j < \lambda/2$ and $\delta_j > \lambda$, it is selected as a source node. The migrated region has loads α_i' and β_i' with $|\alpha_i' - \beta_i'| \leq \lambda/2$, where the critical dimension (higher load) dominates. After migration, d_j remains less than $\lambda/2$, and the node's load decreases by at most $\lambda/2$. If the initial load is $L > 1+\lambda$, the new load satisfies:

$$L - \alpha_i' \geq L - \lambda/2 > 1+\lambda-\lambda/2 = 1+\lambda/2. \qquad (5)$$

  Thus, the node's load remains above $1+\lambda/2$. Through repeated migrations, the load eventually decreases to $\max(A_j', B_j') \leq 1+\lambda$, achieving balance and no longer participating in the scheduling process.
- **Case 2:** $d_j \geq \lambda/2$**.** If a node has $d_j \geq \lambda/2$ and $\delta_j > \lambda$, it may act as a source or target node. In either case, d_j decreases. Since $|\alpha_i' - \beta_i'| \geq x$ (where x is the lower bound of $|\alpha_i' - \beta_i'|$ across all regions with active workloads), migrating this region reduces d_j by at least x, until the node transitions to Case 1. The node's load remains above $1+\lambda/2$ until balanced. Other nodes may remain with loads below $1+\lambda/2$ or transition to Case 2 after migration. All nodes eventually reach a balanced state.

Let $Y = \Delta - 1$ be the maximum load excess above 1 for any node. For Case 1, the number of migrations to reduce the load to $1+\lambda$ is at most Y/x. For Case 2, the number of migrations to reduce d_j to below $\lambda/2$ is at most d_j/x. Thus, the algorithm can converge to a balanced state in a finite number of steps.

Load distribution estimation. Accurate load-based splitting requires the knowledge of the load distribution of a region. Libra estimates the load distribution by sampling requests over a pre-specified period (e.g., 10 s based on our empirical measurement), which enables us to capture sufficient requests for hot regions based on the temporal locality of load distribution. As shown in Figure 7, when splitting a region, the node records the details of all processed requests for the region over a pre-specified period, including the *start_key*, *end_key*, and data volume. For each given key, Libra finds the cumulative load of requests whose keys are smaller than the given key and divides it by the total load to obtain a load ratio, so as to establish a functional relationship between keys and the load distribution. This enables precise load-based splitting. Note that splitting is a one-time operation, and the collected information can be removed after splitting.

Bound-constrained splitting. Load-based splitting divides a region to achieve a target load ratio, so as to ensure that new regions meet load requirements (i.e., loads of partitioned regions are at most $\lambda/2$). Based on the ordering of KV pairs, Libra can accumulate request loads in key order until the desired load ratio is reached, and mark the current key as the split point. However, for requests with overlapping keys, finding the exact split point becomes complicated and potentially leads to inaccurate load splitting.

Libra employs a heuristic using upper and lower bounds to find an approximate split point. Figure 7 depicts the idea, in which a two-way arrow represents the scope of a request in the key space from the *start_key* to the *end_key*. When the split ratio is, say τ, Libra sorts requests by *start_key* in ascending order and accumulates request loads until the cumulative load reaches τ. The *start_key* of the next request defines the minimum split point (lower bound), so that the left-side load is at most τ. Next, Libra sorts requests by *end_key* in descending order and accumulates loads until reaching $1-\tau$. The *end_key* of the current request defines the maximum split point (upper bound), with the left-side load at least τ. Finally, the midpoint of the lower and upper bounds serves as the split point. This heuristic works for both overlapping and non-overlapping requests. For overlapping requests, it approximates the split point. As the overlap diminishes, the upper and lower bounds converge, leading to a more accurate estimate. Without overlaps, the heuristic provides accurate results.

C. Scheduling Plan Optimization

To optimize scheduling, Libra adopts a *lazy scheduling* strategy by deferring region splitting and migration until necessary and batching operations to mitigate overhead. This avoids unnecessary migration. As shown in Figure 8, the process can be decomposed into three stages.

- *Stage 1: Queue maintenance.* The manager maintains a dedicated region queue for each node and stores region load information.
- *Stage 2: Candidate selection.* When Libra triggers scheduling for a hot node, it sorts the regions of the hot node by the critical dimension's load in descending order. Libra evaluates each region sequentially. If the region load exceeds $\lambda/2$ and no suitable destination node is found, it is added to the *split candidate queue*. Otherwise, it is added to the *schedule candidate queue* if the region load is smaller than $\lambda/2$ or the destination node satisfies: the load after migration will not exceed $1+\lambda$, and (ii) the load difference between two dimensions does not increase.
- *Stage 3: Execution.* There are two possible cases. In Case 1 (Stage 3-1), if scheduling requirements are not met but the accumulated load in the split candidate queue is significant, the *schedule candidate queue* is cleared, and splitting proceeds. In Case 2 (Stage 3-2), if the regions in the schedule candidate queue reduce the node's load below one, Libra clears the *split candidate queue* and proceeds with scheduling and performs migration. All cleared operations are not executed.

Optimization analysis. Libra's lazy scheduling involves three optimizations. First, Libra adopts deferred splitting, in which regions not involved in scheduling or involved in the schedule candidate queue are not split, even if their load exceeds λ. This avoids unnecessary splits when the resulting regions can be scheduled on the same node or when balance can be achieved by scheduling other unsplit regions. Second, Libra adopts efficient migration, which prevents inefficient migrations of small-load regions in the schedule candidate queue and prioritizes splits

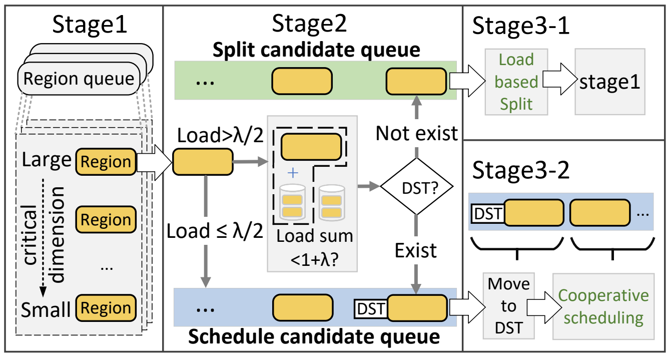


Figure 8: Lazy scheduling.

for better load distribution. Finally, Libra avoids generating too many regions by merging small, cold regions in the background at a low rate as in production [4], [8], [52], to minimize the impact on performance.

V. Evaluation

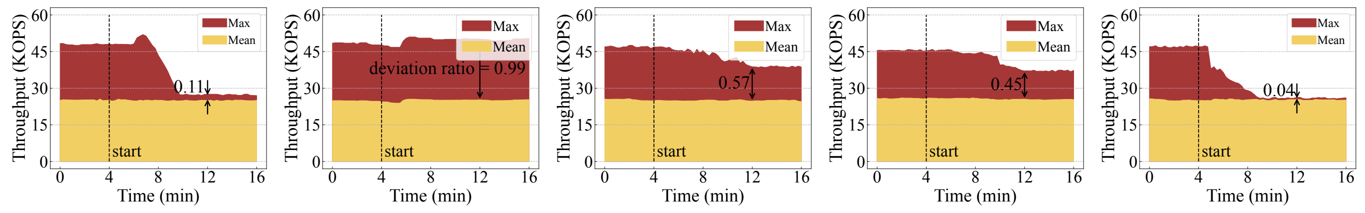
We implement Libra on TiKV [52] and compare it with multiple scheduling strategies. We evaluate the load-balancing effectiveness (§V-B), microbenchmark performance (§V-C), and performance under YCSB workloads [7], as well as the impact of Libra's components (§V-E) and configurations (§V-F).

A. Experiment Setup

Testbed. We conduct experiments on an in-house cluster comprising five physical machines, each equipped with a 20-core 2.1 GHz Intel Xeon Gold 5218R CPU, 32 GB of RAM, and a 1 TB SATA SSD, running Ubuntu 22.04.5 LTS. All machines are interconnected by a 10 Gbps network. We set up nine nodes: one manager node and eight data nodes. One machine hosts the manager node, and each of the remaining four machines hosts two data nodes, with eight cores allocated per data node. We use TiKV [52] (v5.4.0), a production KV store using range-based sharding [11], [14], [58], as the codebase to ensure fair comparisons across various scheduling policies.

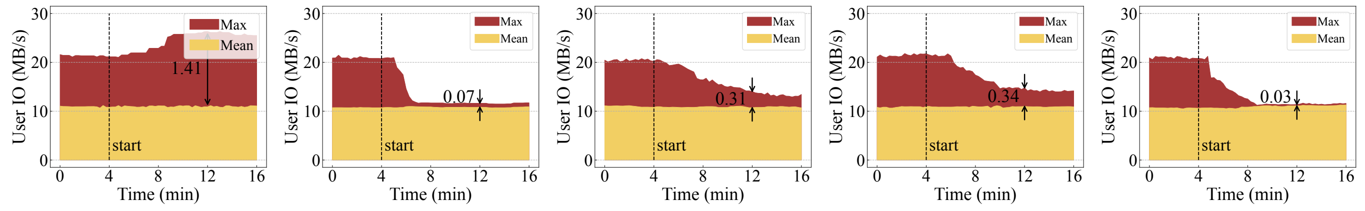
Workloads. We use the extended YCSB benchmark (based on go-YCSB [51] v1.0.1), with the Mixgraph hotness distribution model [9] to better simulate KV access patterns in real-world production environments (except for the experiments in §V-D, which use the native YCSB workload). Also, we follow the common practice of configuring the KV pair sizes to follow a generalized Pareto distribution [27]. We generate two types of workloads: (i) *discrete workloads*, which include discrete operations (e.g., reads, writes, and scans) for microbenchmarks; and (ii) *mixed workloads*, which use YCSB's core workloads A-F for evaluation in practical settings.

Baselines. We compare Libra against three representative scheduling strategies, namely single-dimensional (1D-CPU and 1D-IO), two-dimensional isolated (2D-ISO), and two-dimensional weighted scheduling (2D-WTD), as detailed in §II-C. The weight for 2D-WTD is set to a ratio of 0.5:0.5 (see §V-F for details). All strategies are implemented within TiKV to ensure a fair comparison and are extended to support the *metric mapping* in §IV-A for CPU and disk I/O scheduling.



(a) 1D-CPU (b) 1D-IO (c) 2D-ISO (d) 2D-WTD (e) Libra

Figure 9: Throughput distribution after deploying different approaches.



(a) 1D-CPU (b) 1D-IO (c) 2D-ISO (d) 2D-WTD (e) Libra

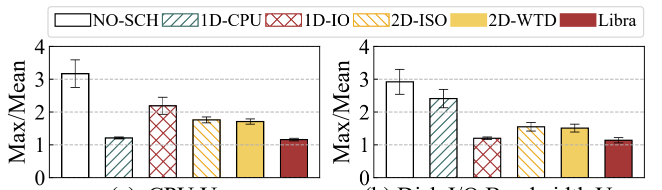
Figure 10: User I/O distribution after deploying different approaches.

Configurations. To capture hotness skewness, the hotness distribution in Mixgraph follows a probability density function $f(x) = ae^{bx} + ce^{dx}$ [21]. To reflect real workload characteristics observed in Facebook's production clusters, we follow the default parameter setting ($a = 14.18$, $b = -2.917$, $c = 0.0164$, and $d = -0.08082$) [21]. We set $\lambda = 0.05$ (see the rationale explained in §V-F), meaning that the scheduling will be triggered when the maximum load exceeds 1.05× the average load across all nodes. Each region has three replicas for fault tolerance. For each experiment, we show the average results over five runs, with error bars indicating the standard deviation.

B. Load Balancing Effectiveness

Load distribution. Figures 9 and 10 depict the load distribution among servers. The red portion represents the load of the most heavily loaded node (Max), while the yellow portion indicates the average load across all nodes (Mean). Figure 9 presents the OPS corresponding to CPU load under different scheduling policies, while Figure 10 shows the user I/O bandwidth usage, reflecting the consumption of disk I/O bandwidth. We activate all scheduling policies at the 4th minute. Until the 16th minute, all policies have completed their scheduling tasks, and the load remains stable. We quantify the difference between Max and Mean (calculated as $(MAX - Mean)/Mean$). A smaller value indicates a higher degree of balance.

By comparing the results in Figures 9 and 10, we find that Libra achieves balanced performance in both dimensions simultaneously and attains a significantly smaller Max-Mean gap than all baselines. As shown in Figures 9(a) and 10(a), 1D-CPU effectively balances the load in the CPU dimension, but fails to balance the other; 1D-IO exhibits a similar pattern. For the two-dimensional schedulers (i.e., 2D-ISO and 2D-WTD) (see Figures 9(c), 9(d), 10(c), and 10(d)), although they partially mitigate imbalances in both CPU and disk I/O loads, they exhibit a notable Max-Mean gap, indicating that their load balancing effectiveness remains limited.



(a) CPU Usage (b) Disk I/O Bandwidth Usage

Figure 11: The degree of balance in two dimensions.

Load balance metric. To quantify the effectiveness of Libra on load balancing, we use the *Max/Mean ratio* (the maximum load divided by the average load) to measure the degree of load balance. A lower ratio implies a higher degree of balance, with the ideal value being one (i.e., all nodes have the same load). For brevity, we present results under a write-only workload, while results under other workloads show similar trends. We fix the request rate for all scheduling strategies for fair comparisons. To obtain the CPU and disk I/O utilizations, we first perform scheduling until all migration operations have been completed. We then run the system for another 10 minutes and sample the CPU and disk I/O utilizations every 15 seconds to obtain average results. This provides sufficient sampling points to ensure accuracy of the results.

Figure 11 shows that Libra consistently achieves the best performance in balancing both CPU and disk I/O loads among all scheduling policies. Compared to 1D-CPU and 1D-IO, Libra reduces the Max/Mean ratio by 4.1% and 47.0% in CPU, and 52.7% and 5.0% in disk I/O, respectively. While either 1D-CPU or 1D-IO achieves performance similar to Libra in CPU or disk I/O, they fail to balance both CPU and disk I/O simultaneously. Compared to 2D-ISO and 2D-WTD, Libra reduces the Max/Mean ratio by 34.1% and 32.2% in CPU, and by 26.5% and 24.5% in disk I/O, respectively. The performance advantage of Libra comes from the innovation in scheduling principle. While 2D-WTD and 2D-ISO consider the CPU and disk I/O dimensions in isolation or at the expense of

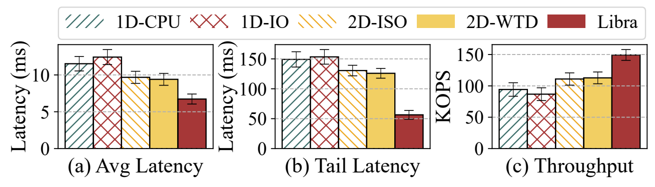


Figure 12: Write performance.

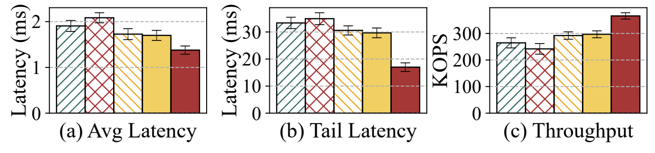


Figure 13: Read performance.

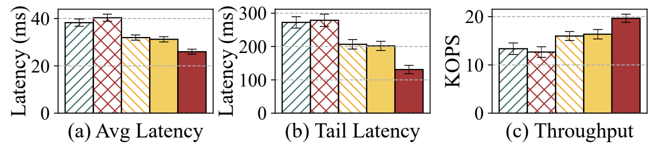


Figure 14: Scan performance.

compromising the accuracy of load status, Libra cooperatively considers both dimensions based on precise load information.

C. Microbenchmarks

We evaluate write, read, and scan operations, and report average latency, 99th-percentile (P99) latency, and throughput for different scheduling strategies. For each run, we send 10 million requests to ensure statistical accuracy.

Write performance. Figure 12 shows the write performance. For latency, compared to 1D-CPU, 1D-IO, 2D-ISO, and 2D-WTD, Libra reduces the average latency by 41.6%, 45.8%, 30.4%, and 28.4%, and reduces the P99 tail latency by 62.1%, 63.2%, 56.7%, and 55.1%, respectively. Thus, Libra responds to upper-layer applications much more promptly and improves the responsiveness of KV storage. For throughput (Figure 12(c)), Libra achieves 32.4%-72.1% higher throughput.

Read performance. Figure 13 shows the read results. Compared to 1D-CPU, 1D-IO, 2D-ISO, and 2D-WTD, Libra reduces the average latency by 27.7%, 33.9%, 20.2%, and 18.9%, and the P99 tail latency by 49.0%, 51.3%, 44.4%, and 42.7%, respectively. It also achieves 23.3%-51.3% higher read throughput.

Scan performance. Figure 14 shows the scan results. Compared to 1D-CPU, 1D-IO, 2D-ISO, and 2D-WTD, Libra reduces average latency by 32.1%, 35.6%, 18.7%, and 16.8%, and the P99 tail latency by 51.9%, 53.0%, 36.6%, and 35.1%, respectively. It also achieves 20.2%-55.3% higher throughput.

Summary. Libra shows significant performance improvements for all operations. Notably, it reduces tail latency by 35.1%-63.2%, effectively avoiding CPU and disk I/O bottlenecks.

D. YCSB Evaluation

YCSB is the standard for evaluating KV stores and comprises six workloads, each representing a typical real-world scenario. Thus, we evaluate Libra using the six core workloads: A (50% reads, 50% updates), B (95% reads, 5% updates), C (100% reads), D (95% reads, 5% inserts), E (95% scans, 5% inserts), and F (50% reads, 50% read-modify-writes). In each workload, we send 10 million operations, a sufficient number of operations to ensure stable experimental results. The only exception is Workload E, which is limited to one million operations, as Workload E is scan-dominated. We reduce the number of operations to control the total experiment time.

Figure 15 presents the results, with all values normalized to 2D-ISO. The results demonstrate that Libra is effective across diverse workloads. In Figure 15(a), Libra shows an average latency reduction of 26.0%, 17.3%, 22.8%, 18.3%, 17.7%, and 26.4% for workloads A to F, respectively. In Figure 15(b), Libra shows a 35.7%-52.7% reduction in the P99 tail latency for all workloads. In Figure 15(c), Libra achieves 22.1%-33.6% improvements in throughput. In Figure 15(d), Libra demonstrates significant improvements in load balancing, achieving a 20.9%-35.8% reduction in the Max/Mean ratio.

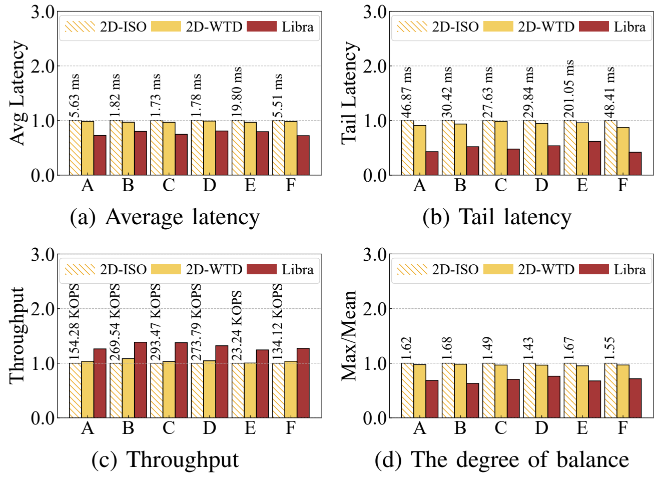


Figure 15: YCSB performance. The numbers above the bars represent the values of the corresponding indicators of 1D-ISO.

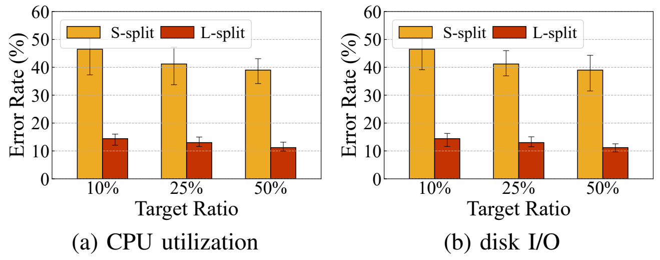


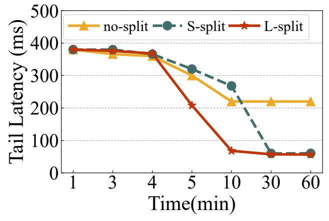
Figure 16: Load splitting accuracy.

Table I: Rate and CPU usage of heartbeats.

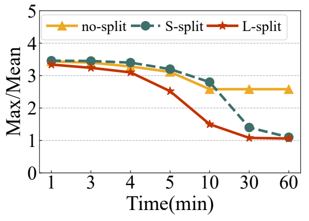
Type	Rate (per node)	CPU usage
Store heartbeat	6 / min	<1%
Region heartbeat	100 $\sim$ 200 / s	5$\sim$10%

E. Performance Breakdown

Load monitoring. To evaluate the overhead of load monitoring, we measure both types of heartbeat messages and present the results in Table I. Both types cause less than 10% additional CPU usage, demonstrating the framework's lightweight feature.

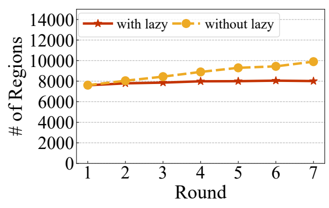


(a) Tail latency

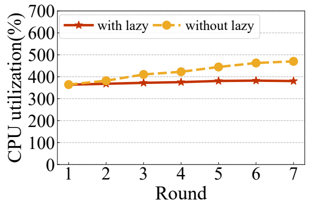


(b) Degree of balance

Figure 17: Impact of the splitting policy.

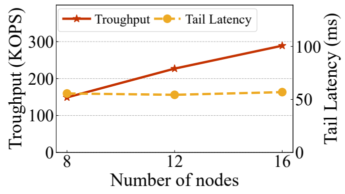


(a) Number of regions

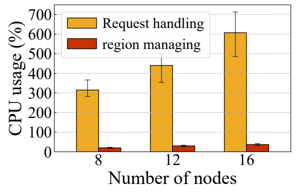


(b) Average CPU usage

Figure 18: Impact of lazy scheduling.

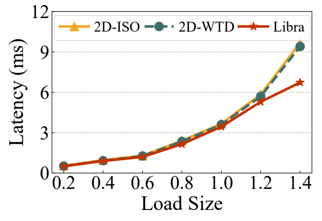


(a) Performance

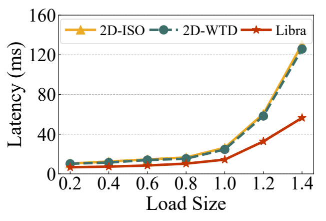


(b) Manager node overhead

Figure 19: Impact of number of nodes.



(a) Average latency



(b) Tail latency

Figure 20: Impact of load traffic.

Load-based splitting. To evaluate the accuracy of load-based splitting (L-split), we randomly select 500 regions for splitting and compare the results against size-based splitting (S-split). Figure 16 presents the average results. The x-axis represents the target load ratio for the newly split region, while the y-axis shows the *error rate*, calculated by dividing the difference between the target and actual loads by the target load. In Figure 16(a), L-split consistently achieves significantly higher accuracy for CPU utilization, with an error rate of only about 10%. In contrast, S-split exhibits a high error rate of more than 40%. Figure 16(b) shows a similar pattern for disk I/O, indicating that L-split consistently achieves higher accuracy.

To evaluate the impact of L-split on system performance, we compare the performance of Libra under three splitting strategies: no-split (no splitting), S-split, and L-split. Figure 17 illustrates the variations in latency and the Max/Mean ratio under these three policies during scheduling. L-split achieves the highest degree of load balance with the fastest scheduling speed, as it accurately partitions regions that require scheduling. Although S-split also attains load balancing, it requires more splitting and scheduling, taking 20 minutes longer than L-split to reach a balanced state. Without splitting, the system fails to achieve load balancing due to overloaded regions.

Lazy scheduling. To evaluate the effectiveness of lazy scheduling, we compare the performance of Libra with and without this mechanism. To examine its long-term behavior, we conduct multi-round tests, each under a different hotspot distribution. We record both the number of regions and the average CPU overhead from internode communication, as shown in Figure 18. The x-axis represents the number of rounds, each corresponding to the scheduling process for a hotspot. Lazy scheduling effectively reduces both the number of regions and the CPU overhead resulting from communication across numerous small regions. In contrast, without lazy scheduling, the number of regions increases rapidly, resulting in significantly higher CPU overhead (an increase of 105.9% after seven rounds).

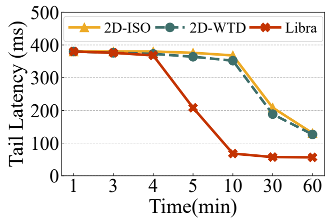
F. *Impact of Configurations*

We study the impact of several key configuration parameters in this section. For brevity, we focus on the write-only workload, as other workloads show qualitatively similar results.

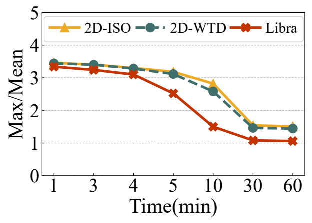
Impact of number of nodes. To evaluate the scalability of Libra, we test its performance across clusters of 8, 12, and 16 nodes, while keeping the per-node configuration unchanged. Figure 19(a) shows that Libra's throughput scales linearly with the number of nodes while latency remains stable, which can be attributed to its robust load balancing across all nodes irrespective of the cluster size. Figure 19(b) shows the CPU overhead of the manager node. The CPU utilization for processing requests grows linearly, scaling with the overall throughput. The overhead of region management, including heartbeat handling and scheduling decisions, increases only marginally, indicating that Libra's scheduling framework imposes minimal overhead and does not hinder the system's inherent scalability. We also develop and open-source a load-balancing simulator that models the algorithm's computation and scheduling costs at any scale, producing results that corroborate our experimental findings.

Impact of load traffic. We evaluate performance by varying the client request rate with different load sizes. Before each test, we form a baseline by running the system without scheduling and record the throughput as load traffic 1. Figure 20 shows the average and tail latency, with the x-axis showing the ratio of each request rate compared to the baseline. Under light loads (i.e., ratio < 1), bottlenecks are rare, resulting in low latency. However, under high loads, 2D-ISO and 2D-WTD have sharp latency increases, while Libra consistently maintains low latency. Compared to 2D-ISO and 2D-WTD, Libra reduces tail latency by 41.5%-56.7% for load ratios ≥ 1. Libra prevents nodes from reaching CPU and disk I/O limits, thereby reducing latency under heavy loads.

Impact of hotspot duration. Figure 21 shows the performance of 2D-ISO, 2D-WTD, and Libra under different hotspot durations. The tail latency and Max/Mean ratio remain largely

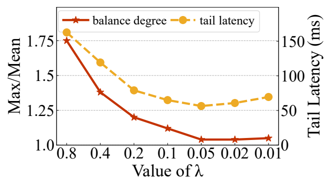


(a) Tail latency

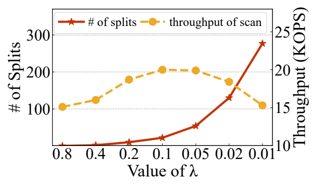


(b) Degree of balance

Figure 21: Impact of hot spot duration.



(a) Performance



(b) Number of splits

Figure 22: Impact of parameter λ.

Table II: 2D-WTD with different weights.

CPU weight	Disk I/O weight	Max/mean	Tail latency (ms)
0	1	2.19	153.32
0.25	0.75	2.04	139.45
0.5	0.5	1.71	125.84
0.75	0.25	2.11	133.22
1	0	2.41	149.10

unchanged for durations under 5 minutes, as migration requires time to complete. After the 5-minute mark, Libra improves steadily and outperforms the others. By the 30-minute mark, Libra achieves load balance, whereas 2D-ISO and 2D-WTD remain engaged in the balancing process. The results indicate that Libra schedules more efficiently than other baselines.

Impact of parameter λ. We evaluate Libra with different values of λ. As shown in Figure 22(a), a smaller λ, which reflects lower tolerance for imbalance, leads to a lower Max/Mean ratio. The improvements in load balance become negligible beyond $\lambda = 0.05$ due to estimation errors and system noise. Additionally, tail latency increases due to more splitting and overhead from managing many small regions. Figure 22(b) shows that once λ exceeds 0.05, the number of splits rises sharply and scan throughput drops by 23.1%, mainly because cross-region scanning overhead outweighs load balancing benefits. Thus, we choose $\lambda = 0.05$ as the default in Libra.

Impact of weight setting in 2D-WTD. To appropriately configure the weights in the 2D-WTD, we evaluate various weight configurations for 2D-WTD to determine their impact, and the results are summarized in Table II. In our experiments, the algorithm performs best at a 0.5:0.5 weight ratio, as both CPU and disk I/O represent critical bottlenecks that require balanced scheduling. The optimal weight may shift in scenarios where hardware configuration changes. For instance, if the CPU is superior while disk I/O remains a bottleneck, assigning a higher weight to I/O can enhance system performance. This highlights a key limitation of 2D-WTD: selecting an effective weight is challenging and the weight lacks adaptability.

VI. Related Work

KV stores. Many KV stores employ the LSM-tree as the storage engine [3], [22], [25], [45], [58]. To improve LSM-tree management, various optimizations have been proposed, including compaction optimization [17], [24], [56], [57], KV separation [10], [37], [45], Bloom filter management [16], [44], learned indexes [15], parameter tuning [16], [17], [62], etc. Recent work also explores LSM-tree designs for new storage devices, such as persistent memory [31], [33], [35]. Distributed KV stores combine multiple KV store instances over a network for high performance and data reliability [3], [18], [48], [52], [58]. A key challenge in such systems is load imbalance. Libra addresses this issue in LSM-tree-based KV stores while remaining compatible with existing LSM-tree optimizations.

Load balancing. Load balancing of distributed systems has been extensively studied. Stateless systems [49] can distribute tasks dynamically without moving data, whereas stateful systems, such as distributed KV stores, depend on data placement. A common approach is random static placement via consistent hashing [2], [34], [38], [55]. However, consistent hashing is unsuitable for systems based on range sharding and cannot handle dynamic load effectively [23]. Some studies leverage caching to support load balancing [23], [30], [42], while others use replication to distribute requests among replicas [41], [47], [64]. However, they are effective only for read-heavy workloads. Libra adopts *migration* by dynamically adjusting the data distribution, offloading hot data from overloaded nodes to underutilized ones [4], [36], [52], [58]. Libra can also work alongside caching and replication.

Migration. Existing distributed KV stores implement migration by moving regions or data within specific ranges, based on software-level metrics [4], [52], [58], [59]. Libra improves load balancing by carefully balancing both CPU utilization and disk I/O. Some studies enable effective hotspot detection for migration [29], [66]; however, they lack the ability to capture region-level hardware resource statistics, a core capability of Libra. Other works focus on optimizing the efficiency of data movement during migration [32], [67], while Libra focuses on the migration decision-making.

VII. Conclusion

We identify and analyze the dual load imbalance in distributed KV stores and propose Libra, a cooperative scheduling framework that balances CPU and disk I/O loads simultaneously and dynamically. Experiments confirm that Libra achieves significant performance enhancements. In future work, we will automate parameter tuning to reduce manual effort. Although Libra is theoretically transferable to heterogeneous environments, we plan to implement the necessary development to make it practical. Finally, we will extend the framework to non-KV systems to demonstrate its general applicability.

Acknowledgments. This work is supported in part by NSFC (62172382, 62472392), Major Science and Technology Project of Fuzhou City (2024-ZD-005) and Research Grants Council of Hong Kong (GRF 14214622). The corresponding author is Yongkun Li.

VIII. AI-Generated Content Acknowledgement

This paper does not contain any AI-generated content. The LLM tools (ChatGPT and Copilot) were used only for grammar checking.

References

[1] Alibaba, "OceanBase," https://github.com/oceanbase/oceanbase, 2024.

[2] Amazon, "DynamoDB," https://aws.amazon.com/dynamodb, 2024.

[3] Apache, "Cassandra," https://github.com/apache/cassandra, 2024.

[4] ——, "HBase," https://github.com/apache/hbase, 2024.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS/PERFORMANCE joint international conference*, 2012.

[6] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing latency spikes in Log-Structured merge Key-Value stores," in *Proceedings of USENIX ATC*, 2019.

[7] brianfrankcooper, "YCSB," https://github.com/brianfrankcooper/YCSB, 2024.

[8] B. Calder, J. Wang, and et al., "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of ACM SOSP*, 2011.

[9] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDBKey-Value workloads at facebook," in *Proceedings of USENIX FAST*, 2020.

[10] C. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling efficient updates in KV storage via hashing," in *Proceedings of USENIX ATC*, 2018.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of USENIX OSDI*, 2006.

[12] T. Chen, M. Li, Y. Li, and et al., "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[13] R. Community, "Riak," https://github.com/basho/riak, 2024.

[14] J. C. Corbett, J. Dean, and et al., "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems*, vol. 31, no. 3, pp. 1–22, 2013.

[15] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "From WiscKey to bourbon: A learned index for Log-Structured merge trees," in *Proceedings of USENIX OSDI*, 2020.

[16] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal Navigable Key-Value Store," in *Proceedings of ACM SIGMOD*, 2017.

[17] N. Dayan and S. Idreos, "Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging," in *Proceedings of ACM SIGMOD*, 2018.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, and et al., "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of ACM SOSP*, 2007.

[19] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Transactions on Storage*, vol. 17, no. 4, pp. 1–26, 2021.

[20] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A Distributed, Searchable Key-Value Store," in *Proceedings of ACM SIGCOMM*, 2012.

[21] Facebook, "RocksDB Trace, Replay, Analyzer, and Workload Generation," https://github.com/facebook/rocksdb/wiki/RocksDB-Trace,-Replay,-Analyzer,-and-Workload-Generation, 2022.

[22] ——, "RocksDB," https://github.com/facebook/rocksdb, 2024.

[23] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *Proceedings of ACM SOCC*, 2011.

[24] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling Concurrent Log-structured Data Stores," in *Proceedings of ACM EuroSys*, 2015.

[25] Google, "LevelDB," https://github.com/google/leveldb, 2024.

[26] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proceedings of ACM SoCC*, 2013.

[27] J. R. Hosking and J. R. Wallis, "Parameter and Quantile Estimation for the Generalized Pareto Distribution," *Technometrics*, vol. 29, no. 3, pp. 339–349, 1987.

[28] D. Huang, Q. Liu, Q. Cui, and et al., "Tidb: a raft-based htap database," *Proceedings of the VLDB Endow*, vol. 13, no. 12, pp. 3072–3084, 2020.

[29] H. Jin, Z. Li, H. Liu, X. Liao, and Y. Zhang, "Hotspot-aware hybrid memory management for in-memory key-value stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 779–792, 2020.

[30] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of ACM SOSP*, 2017.

[31] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. R. Choi, "SLM-DB: Single-Level Key-Value Store with Persistent Memory," in *Proceedings of USENIX FAST*, 2019.

[32] J. Kang, L. Cai, F. Li, X. Zhou, W. Cao, S. Cai, and D. Shao, "Remus: Efficient live migration for distributed databases with snapshot isolation," in *Proceedings of ACM SIGMOD*, 2022.

[33] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for Nonvolatile Memory with NoveLSM," in *Proceedings of USENIX ATC*, 2018.

[34] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proceedings of ACM STOC*, 1997.

[35] W. Kim, C. Park, D. Kim, H. Park, Y. ri Choi, A. Sussman, and B. Nam, "ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory," in *Proceedings of USENIX OSDI*, 2022.

[36] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper, "The yahoo! cloud datastore load balancer," in *Proceedings of ACM CloudDB*, 2012.

[37] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's Key-value Storage System for Cloud Data," in *Proceedings of IEEE MSST*, 2015.

[38] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[39] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *Proceedings of ACM SOSP*, 2019.

[40] H. Li, S. Jiang, C. Chen, A. Raina, X. Zhu, C. Luo, and A. Cidon, "RubbleDB:CPU-Efficient replication with NVMe-oF," in *Proceedings of USENIX ATC*, 2023.

[41] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports, "Pegasus: Tolerating skewed workloads in distributed storage with In-Network coherence directories," in *Proceedings of USENIX OSDI*, 2020.

[42] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with SwitchKV," in *Proceedings of USENIX NSDI*, 2016.

[43] Y. Li, Z. Liu, P. P. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui, "Differentiated Key-Value storage management for balanced I/O performance," in *Proceedings of USENIX ATC*, 2021.

[44] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores," in *Proceedings of USENIX ATC*, 2019.

[45] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating Keys from Values in SSD-Conscious Storage," in *Proceedings of USENIX FAST*, 2016.

[46] C. Luo and M. J. Carey, "On performance stability in lsm-based storage systems (extended version)," *arXiv preprint arXiv:1906.09667*, 2019.

[47] M. MITZENMACHER, "The power of two random choices: a survey of techniques and results," *Handbook of Randomized Computing*, vol. 1, pp. 255–312, 2001.

[48] R. Nishtala, H. Fugal, and et al., "Scaling memcache at facebook," in *Proceedings of USENIX NSDI*, 2013.

[49] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *Proceedings of USENIX NSDI*, 2018.

[50] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[51] PingCAP, "go-ycsb," https://github.com/pingcap/go-ycsb/tree/v1.0.1, 2024.

[52] ——, "TiKV," https://github.com/tikv/tikv, 2024.

[53] M. Qin, Q. Zheng, J. Lee, B. Settlemyer, F. Wen, N. Reddy, and P. Gratz, "Kvrangedb: Range queries for a hash-based key–value device," *ACM Transactions on Storage*, vol. 19, no. 3, pp. 1–21, 2023.

[54] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *Proceedings of ACM SOSP*, 2017.

[55] Redislab, "Redis," https://github.com/redis/redis, 2024.

[56] R. Sears and R. Ramakrishnan, "bLSM: A General Purpose Log Structured Merge Tree," in *Proceedings of ACM SIGMOD*, 2012.

[57] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building Workload-Independent Storage with VT-Trees," in *Proceedings of USENIX FAST*, 2013.

[58] R. Taft, I. Sharif, and et al., "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of ACM SIGMOD*, 2020.

[59] N. VanBenschoten, A. Ajmani, and et al., "Enabling the next generation of multi-region applications with cockroachdb," in *ACM SIGMOD*, 2022.

[60] Y. Wang, C. Li, X. Shao, Y. Chen, F. Yan, and Y. Xu, "Lunule: an agile and judicious metadata load balancer for cephfs," in *SC*, 2021.

[61] T. Yao and Y. e. a. Zhang, "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proceedings of USENIX ATC*, 2020.

[62] J. Yu, S. H. Noh, Y. ri Choi, and C. J. Xue, "ADOC: Automatically harmonizing dataflow between components in Log-StructuredKey-Value stores for improved performance," in *Proceedings of USENIX FAST*, 2023.

[63] Q. Zhang, Y. Li, P. P. C. Lee, Y. Xu, Q. Cui, and L. Tang, "Unikv: Toward high-performance and scalable kv storage in mixed workloads via unified indexing," in *Proceedings of IEEE ICDE*, 2020.

[64] Q. Zhang, Y. Li, P. P. Lee, Y. Xu, and S. Wu, "DEPART: Replica decoupling for distributed Key-Value storage," in *Proceedings of USENIX FAST*, 2022.

[65] T. Zhang, J. Wang, and et al., "FPGA-Accelerated compactions for LSM-based Key-Value store," in *Proceedings of USENIX FAST*, 2020.

[66] Z. Zhou, J. Zu, E. Huang, Z. Wang, H. Zhou, D. Zhang, X. Chen, and C. Wu, "Spotmon: Enabling general hotspot monitoring in key-value stores," in *Proceedings of IEEE ICNP*, 2024.

[67] Z. Zhu, Y. Zhao, and Z. Liu, "In-Memory Key-Value store live migration with NetMigrate," in *Proceedings of USENIX FAST*, 2024.